1st Asian-Pacific Summer School on Formal Methods
Class 11-12: Deductive verification with Frama-C

1 Introduction

The goal of these classes is to prove with Frama-C and its jessie plugin that a C implementation of insert sort is correct. The C code itself is in `insert_sort.c`. The specification, helper annotations and Coq proofs will be built during the classes.

The algorithm work as follows. We first have a function which insert a new value v in an already sorted array a of length l (provided $a[l]$ is a valid location). For that, we start at index $l-1$ and shift each cell to the right until we reach an index i for which $a[i] <= v$: v can then be inserted at index $i+1$.

The sort function itself inserts each element of the original array one by one, so that at each step i, the sub-array of length i is sorted.

A C implementation is the following:

```
void insert(int* arr, int length, int val);

void insert_sort(int* arr, int length);

void insert(int* arr, int length, int val) {
  int i = length −1;
  while(i≥0 ∧ arr[i] > val) { arr[i+1] = arr[i]; i−−; }
  arr[i+1] = val;
  return;
}

void insert_sort(int* arr, int length)  {
  for(int i = 1; i < length; i++) { insert(arr,i,arr[i]); }
}
```

The properties that we want to establish are the following:

- safety: all array accesses are valid
- sort: at the end of the `insert_sort` function, the array `arr` is sorted
- elements: at the end of `insert_sort`, `arr` contains the same elements as before

2 Logic Specifications

Before writing the function contracts, we have to define the predicates that we will use.

1. Define a predicate `sorted` which, given an array and its length is true iff the value of each cell of the array is less than or equal to the value of the next one.

2. (Optional) Write a lemma that says that this predicate is equivalent to the predicate seen during the course.

3. In order to ensure that the array always contain the same elements, we will use a function `nb_occ` that counts the number of occurrences of a given value in an array. This function can be defined axiomatically:

   - the function takes as input an array, the two bounds between which we count the occurrences, and the value to be searched
   - a first axiom indicates that if the higher bound is less than the lower one, then `nb_occ` is 0
   - if the bounds *low* and *high* are in the correct order, there are two cases (thus two axioms): the number of occurrences from from *low* to *high* is equal to the number of occurrences from *low* to $high - 1$ if the last cell does not contain the value of interest. Otherwise, we have to add one to this number.

3 Function Contracts

3.1 `insert_sort`

The main function of the program is `insert_sort`. We thus start by writing its specification. This is the main property that we want to establish.

1. What are the pre-conditions for the function (validity of the pointers, possible values for the length)?

2. What are the post-conditions (for the sort and elements properties respectively)?

3.2 `insert`

This function is an auxiliary function. Its specification must be complete enough to allow us to prove the contract of `insert_sort`.

1. What are the pre-conditions (validity, possible values of length, fact that the array is sorted)?

2. What are the locations that may be assigned by the function?

3. What is the post-condition for the sort property?

4. What is the post-condition for the elements property? (Don't forget that we are *inserting* an element, so the number of occurrences is not the same for all values).

4 Loop invariants

Loop invariants are not part of the specification *per se*, but it is not possible to perform a proof without them. For both loops, the invariants must in particular indicate the following things

- interval of variation of the indices
- locations that are unchanged
- number of occurrences of the various elements

For `insert`, it is in addition important to state that all cells seen so far are greater than `val`, and that they have been shifted on the right.

For `insert_sort`, the main invariant is that the beginning of the array is sorted.

5 Proof

Once you're confident with your contracts and your invariants, you may try to have alt-ergo prove the program. For that, the command line

```
frama-c -jessie insert_sort.c
```

will launch the gwhy interface with all proof obligations generated by the tool chain. You can then try to have alt-ergo discharge automatically all of them. If this works, you're done. Otherwise, you have to check the failed formulas to see if you must refine your specification (e.g. an invariant is too weak to prove the post-condition, or too strong and does not hold), or if alt-ergo needs some more help.

In particular, a few lemmas about `nb_occ` might be useful:

1. split: if we take a third index `med` between `low` and `high`, the number of occurrences between `low` and `high` is the sum of the occurrences between `low` and `med` and between `med+1` and `high`
2. copy: if a portion (between indices `low2` and `high2` of an array in a state `L2` is a copy of another portion of the array in a state `L1` (between `low1` and `high1`), the number of occurrences between `low2` and `high2` in state `L2` is the same as the number of occurrences between `low1` and `high1` in state `L1`
3. reverse: the axioms give a recursive definition by defining the number of occurrences between `low` and `high` in terms of the number of occurrences between `low` and `high-1`, but we can establish that there is also a relation with the number of occurrences between `low-1` and `high`

In any case, we need Coq to complete the missing proofs. The following command will instruct Frama-C to generate a Coq file in which the "proofs" consists in a call to ergo:

```
frama-c -jessie -jessie-atp coq \
        -jessie-why-opt="--coq-tactic ergo" insert_sort.c
```

In addition, Frama-c launches `coqc` which will fail at the first proof obligation that alt-ergo cannot discharge. You must then edit the generated Coq file `insert_sort_why.v`, which is in directory `insert_sort.jessie/coq/`[1] with coqide or Proof General and replace the calls to alt-ergo that fail by a real Coq proof.

Note that calling alt-ergo through Coq is less efficient than through gwhy. You might want to give alt-ergo a bit more time by writing

```
Dp_timeout 20.
```

at the beginning of the Coq file.

Since writing a Coq proof can be quite time consuming, be really sure that your specification is correct before that.

[1] Don't forget to include the directory in Coq's search path. You can also cd to that directory.